Applications of Linear Types

Dhruv C. Makwana
Trinity College College

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the degree of
Master of Engineering in Part III of the Computer Science Tripos

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: dcm41@cam.ac.uk

May 8, 2018

Declaration

I Dhruv C. Makwana of Trinity College College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 0

Signed:

Date:

TO DO: Abstract

This is the abstract. Write a summary of the whole thing. Make sure it fits in one page.

Contents

List of Figures

List of Tables

1 | TO DO: Introduction

This is the introduction where you should introduce your work. In general the thing to aim for here is to describe a little bit of the context for your work — why did you do it (motivation), what was the hoped-for outcome (aims) — as well as trying to give a brief overview of what you actually did.

It's often useful to bring forward some "highlights" into this chapter (e.g. some particularly compelling results, or a particularly interesting finding).

It's also traditional to give an outline of the rest of the document, although without care this can appear formulaic and tedious. Your call.

In this thesis, I will argue that linear types are an appropriate, *type-based formalism* for the problem of *efficient* matrix-expression compilation. I will show that framing the problem using linear types can help *reduce bugs* by making precise and explicit the informal, ad-hoc practices typically employed by human experts and linear algebra *compilers* and automate checking them. As evidence for this argument, I will show programs written with this safety, precision and explicitness (1) can be just as pleasant and convenient for a programmer as less efficient, but higher-level linear algebra libraries and (2) perform just as *efficiently and predictably* as lower-level, less readable and more error-prone linear algebra libraries.

1.1 Overview of Problem

1.2 Contributions

2 | Background

In this chapter, I will outline the concept of linear types and show how they can be used to solve the problems faced by programmers writing code using linear-algebra libraries. I will be emphasising the *practical* and intuitive explanations of linear types to keep this thesis accessible to working programmers as well as academics not familiar with type-theory; giving only a terse overview of the history and theory behind linear types for the interested reader to pursue further.

2.1 Tracking Resources with Linearity

Familiar examples of using a type-system to express program-invariants are existential-types for abstraction and encapsulation, polymorphic types for parametricity and composition (a.k.a generics). Less-known examples include dependent-types (contracts or pre- and post-conditions). The advantages of using a type-system to express program invariants are summarised by saying the stronger the rules you follow, the better the guarantees you can get about your program, *before* you run it. At first, the rules seems restrictive, but similar to how the rules of grammar, spelling and more generally writing help a writer make it easier and clearer to communicate the ideas they wish to express, so too do typing rules make it easier to communicate the intent and assumptions under which a program is written. An added, but often overlooked benefit is automated-checking: a programmer can boldly refactor in certain ways and the compiler will *assist* in ensuring the relevant invariants the type-system enforces are updated and kept consistent by pointing-out where they are violated.

Linear types are a way to help a programmer track and manage resources. In practical programming terms, they enforce the restriction that a value may be used exactly once.[1] While this restriction may seem limiting at first, precisely these constraints can be used to express common invariants of the programs written by working programmers every day. For example: a file or a socket, once opened *must* closed; all memory that is manually allocated *must* be freed. C++'s destructors and Rust's Drop-trait (and more generally, its borrow-checker) attempt to enforce these constraints by basically doing the same thing: any resource that has not been moved is deallocated at the end of the current lexical scope. Notably, these languages also permit aliasing, alongside rules enforcing when it is acceptable to do so. On face value, the above one-line description of linear types prevents aliasing or functions such as $\lambda x.\ x \times x$, such features are still allowed (albeit in a more restricted fashion) in a *usable* linear type system designed for working with linear-algebra libraries.

[1] This definition may differ from more colloquial uses in discussions surrounding *substructural* type systems and/or Rust.

```
# Numpy (Python)
import numpy.matlib
a = [[1,0],[0,1]]
b = [[4,1],[2,2]]
c = numpy.matmul(a,b)
# Julia
c = [1 0; 0 1] * [4 1; 2 2]
(* Owl (OCaml) *)
open Owl
let a = Mat.of_arrays [| [| 1.; 0. |]; [| 0.; 1. |] |]
let b = Mat.of_array [| [| 4.; 1. |]; [| 1.; 2. |] |]
let c = Mat.(a *@ b)
```

Figure 2.1 – Matrix Multiplication in Numpy (Python), Julia and Owl (OCaml).

2.2 Problem in Detail

Given this background, the most pertinent question at hand is: what problems do linear-algebra library users (and writers) typically face? The answer to this question depends on which of two buckets a programmer falls (or is forced by domain) into. On one side, we have users of high-level linear-algebra libraries such as Owl (for OCaml), Julia and Numpy (for Python); other the other, we have users of more manual, lower-level libraries such as BLAS (Basic Linear Algebra Subroutines) for languages like C++ and FORTRAN.[2] Most of what follows applies to *dense* linear-algebra computations rather than *sparse* because memory allocated for results typically depends on the sparsity of the inputs and so is not immediately amenable to the techniques proposed in this thesis.

2.2.1 One Too Many Copies and a Thousand Bytes Behind

In Figure 2.1, we see that matrix-multiplication is fairly trivial to write and execute in Numpy, Julia and Owl. Let us call this approach *value-semantic*, meaning that objects are *values* just like integers and floating-point numbers. This approach confers two key advantages to the programmmer: it is easy to read (equational and algebraic declarations) and it is easy to reason about (as one would with a mathematical formula). Although this approach does permit *aliasing*, the conse-

[2]I am not including Rust in this comparison because its linear-algebra libraries are under active development and not as well-known/used. Later on, given that it is a language with in-built support of substructural features to track resources, Rust will be compared and contrasted with this project to evaluate the classic (E)DSL-versus-language-feature debate as it applies to the domain of linear-algebra libraries.

```
let mul x y =
  if same_shape x y then
    let y = copy y in
    (_owl_mul (kind x) (numel x) x y y; y)
  else
    broadcast_op (_owl_broadcast_mul (kind x)) x y
```

Figure 2.2 – Implementation of Matrix Multiplication in Owl (OCaml). Note the 'copy' for the result and the unsafe '_owl_mul' operation used to perform an in-place multiplication.

quences are benign because the result of any computation is a *new* value, distinct from any used during the calculation of that value.

However, these advantages come with some costs: constantly producing new values is wasteful on memory (although the example given in Figure 2.1 is only a 2×2 matrix, many real-world datasets can contain up to gigabytes of data). A complex expression may create many short-lived temporaries which would need to be reclaimed by a garbage-collector (see Figure 2.2). Libraries taking a *value-semantic* approach offer a dichotomy for a user wishing to implement a new algorithm: either use the existing and safe primitives to build an easy to reason about but slower, more memory-intensive algorithm, or use escape-hatches (typically provided by most libraries, which permit in-place modification of objects) to build faster, and more efficient algorithms which are harder to reason about.

2.2.2 IHNIWTLM

The title of this subsection[3] illustrates the problem with the C++/FORTRAN side: legibility (and ease of reasoning) is sacrificed at the altar of performance and efficiency.

Although escape-hatches do exist in value-semantic libraries, their use is discouraged. Systematic consideration of performance requires lowering the level of abstraction a programmer is working on. At this level, several factors such as memory layout, allocation, re-use as well as cache behaviour and parallelism become apparent. Of these, memory allocation and re-use are of most relevance to linear-types and this thesis.

In Fortran (Figure 2.3), data is typically allocated statically (at compile time) so temporary storage for all intermediate values must be managed by the program-

[3] I Have No Idea What Those Letters Mean.

```
      program blasMatMul
      implicit none
      real*4 a(2,2), b(2,2), c(2,2)
C     External from BLAS
      external dgemm
C     Initialize in column major storage of Fortran
      data a/ 1,0,
     *        0,1/
      data b/ 4,1,
     *        1,2/
C                tfm  tfm  rowA colB K  alpha a lda  b  ldb beta c  ldc
      call dgemm('N', 'N', 2,   2,   2, 1.0,  a,  2, b, 2,  0.0, c, 2)
```

Figure 2.3 – One of *several* BLAS (Fortran) routines for Matrix Multiplication.

mer. While this approach leads to verbose and less readable code, the explicitness is good for understanding the memory concerns of the program, albeit at the expense of understanding what the program is actually calculating.

On the other hand, C++ (with operator overloading) can end up looking fairly readable. For safety and correctness, expressions are typically handled with value-semantics. However, given *extra* information about, aliasing (Eigen, Figure 2.4) or usage of intermediate expressions (uBLAS, Figure 2.5), the number of temporaries allocated can be reduced and increased *implicitly* to improve performance (remove unnecessary allocations or re-calculations respectively). Further tricks to improve performance include expression templates (building up an expression-tree at compile time and then pattern-matching on it to produce code) and lazy evaluation (only calculating a result when it is needed). These will be discussed in more detail in Chapter 5.

2.3 TO DO: Proposed Solution

A (E)DSL!

2.4 Further Reading and Theory

No exposition of linear types would be complete without a mention of Girard's Linear Logic [2]. As mentioned in the Stanford Encyclopedia of Philosophy, it is "a refinement of classical and intuitionistic logic. Instead of emphasizing truth, as in

```cpp
#include <iostream>
#include <Eigen/Dense>
using namespace std;
int main()
{
    Eigen::Matrix2d a,b,c;
    a << 1, 0,
         0, 1;
    b << 4, 1,
         1, 2;
    c << 0, 0,
         0, 0;
    a * b; // new matrix
    c += a * b; // temporary for correctness in case of aliasing
    c.noalias() += a * b; // no temporaries
}
```

Figure 2.4 – Some examples of Matrix Multiplication in Eigen. Using expression templates (to be discussed later) and *explicitly provided* aliasing information, Eigen can emit a single BLAS 'dgemm'-like call for the last line, mirroring the Fortran example of Figure 2.3.

```cpp
noalias(C) = prod(A, B);
// Preferable if T is preallocated
temp_type T = prod(B,C); R = prod(A,T);
prod(A, temp_type(prod(B,C));
prod(A, prod<temp_type>(B,C));
```

Figure 2.5 – Boost uBLAS example of Matrix Multiplication. Temporaries need to be marked as such to prevent unnecessary re-computation of values.

classical logic, or proof, as in intuitionistic logic, linear logic emphasizes the role of formulas as resources." A walk from logic to programming along the well-trodden Curry-Howard bridge brings us to linear types.

For the category theory inclined reader, the !-operator (sometimes, for reasons elided here, called *exponentiation*) forms a co-monad; for the rest of us, this entails two (rather simple) facts about a value you can use any number of times: you can (1) can use it once (co-unit), and (2) can pass it to many contexts that will use it many times (co-multiply).

More generally, by annotating variables in the context with their usage (when implementing a type-checker for a linearly typed language), we can express the rules of *substructural* (including affine, relevant and ordered type systems) under the more general framework of *co-effects* [3].

Stepping further back, both the practice and theory behind resource-aware programming has made visible progress in the past few years. On the programming side, we have Linear Haskell, Rust and Idris (experimental). On the research side, we have Resource Aware ML [4] and the tantalising promise of integrating linear and dependent types [5].

3 | Implementation

This chapter may be called something else... but in general the idea is that you have one (or a few) "meat" chapters which describe the work you did in technical detail.

I implemented LT4LA in OCaml, however I strongly believe the ideas described in this chapter can be applied easily to other languages and also are modular enough to extend the OCaml implementation to output to different back-end languages. I will show how a small core language with a few features can be the target of heavy-desugaring of typical linear-algebra programs. This core language can then be elaborated and checked for linearity before performing some simple and predictable optimisations and emitting (in this particular implementation) OCaml code that is not obviously safe and correct (with respect to linearity).

3.1 Structure of LT4LA

LT4LA follows the structure of a typical compiler for a (E)DSL. From the start, I made a concerted effort to (1) write pure-functional code (typically using a monadic-style) which helped immensely with modularity and debugging when tests showed errors (2) produce readable, useful and precise error-messages in the hope that someone who did not understand linear types could still use the LT4LA (3) write tests and set-up continuous-integration for all non-trivial functions so that I could spot and correct errors that were not caught by OCaml's type-system whenever I implemented new features or or refactored my code.

1. **Parsing & Desugaring**. A generated, LR(1) parser parses a text file into a syntax tree, which is then desugared into a smaller, more concise abstract syntax tree. The former aims to mimic OCaml syntax with a few extensions and keywords so that it is familiar and thus easy to pick-up for OCaml users. The latter allows for the type-checker to be simpler to implement and easier to specify. In general, this part will vary for different languages or can be dealt with differently using combinators (the EDSL approach) or a syntax-extension if the host language offers such support.

2. **Type-checking**. The abstract syntax tree is explicitly-typed, with some inference to make it less verbose and more convenient to write typical programs.

3. **Matrix Expressions**. During type-checking, if a matrix-expression is encountered, it is either successfully elaborated into an expression in the abstract syntax tree which is then consequently type-checked, or fails to find suitable routines to calculate the given expression.

4. **Code Generation**. The abstract syntax tree is translated into standard OCaml and a few-particular 'optimisations' are made to produce more readable code. This process is type-preserving: the linear type system is embedded into OCaml's type system, and so when the OCaml compiler compiles the generated code, it acts as a sanity check on the code produced.

5. **Executable Artifacts**. A transpiler and a REPL are the main artifacts produced for this thesis.

6. **Tests**. As mentioned before, almost all non-trivial functions have tests to check their behaviour. The output of the transpiler was also tested by having

```
1  let rec f (!i : !int)
2          (!n : !int)
3          (!x0 : !elt)
4          (write : z arr)
5          ('x)
6          (weights : 'x arr)
7          : z arr * 'x arr =
8    if n = i then (write, weights) else
9    let !w0 <- weights[0] in
10   let !w1 <- weights[1] in
11   let !w2 <- weights[2] in
12   let !x1 <- write[i] in
13   let !x2 <- write[i + 1] in
14   let !x : !elt = w0 *. x0 +. w1 *. x1 +. w2 *. x2 in
15   f (i + 1) n x1 (write[i] := x) _ weights
```

Figure 3.1 – A recursive function to perform a *in-place* (simplified) 1D-convolution (sliding weighted-average) using a read-only array of weights and a writeable array, originally containing the input array.

the build system generate code at OCaml code which could then be compiled and tested as usual.

3.2 Core Language

A full description of the core language can be found in Appendix A. To make a linearly-typed language usable, we need some way of using values zero or more than once, as we would an intuitionistic value. For this, we have in the type-expressions the !-constructor and in the term-expressions the **many**-constructor. For soundness, I implemented *value-restriction* for this type system, only *values* (anything not an array, matrix or an expression that can be reduced) that use *no linear values* can be marked with !. This means we can freely have integers, booleans and (appropriate) functions usable as normal. Recursive functions can only use linear values that are passed in as arguments to the function.

So, how does a typical programmer write code that uses a matrix or an array more than once? Figure 3.1 shows an example program that executes a (simplified) 1D-convolution (sliding weighted-average) using a read-only array of three weights. On line 1, `let rec` introduces a recursive function 'f', followed by explicitly annotated formal parameters. Intuitively, `!id` allows `id` to be used intuitionistically. The types `!int` and `!elt` correspond to integers and element types (for example, 64-bit

floating-point).

Array and matrix types are parameterised by *fractional-capabilities*. A fraction of 1 (2^0) represents complete ownership of a value; in particular, this allows a programmer to write or free it. Creating an array gives you ownership of it; the function `array : !int --o z arr` (where `z` represents '0'). Once you have ownership of an array/matrix, you can free it: `free : z arr --o unit`. Importantly, because a linear-value may only be used once, the array just freed is *out of scope* for following expressions, preventing use-after-free. Ownership also enables you to write to the array: `set : z arr --o !int --o !elt --o z arr` (the syntax `w[i] := j` is just sugar for `set w i j`). Here, linearity prevents accessing values which represented the array *before* the mutation.

Any fraction less than 1 (for simplicity, limited to 2^{-k} in this system, for a positive integer k) represents read-only access. So, the `'x` represents a natural number (either a zero `z`, variable `'x` or a successor ($+1$) of a natural number). Hence, you can read from (index) any array `get : 'x . 'x arr --o !int --o !elt * 'x arr` (the syntax `let !v <- w[i]` is just sugar for `let (!v, w) = get _ w i`). Transparently rebinding `w` with the returned value means a program can appear to use `w` multiple times; the underscore is how a programmer tells the compiler to automatically *infer* the correct fractional-capability based on the other arguments passed to the function. Fractions also permit aliasing: `share : 'x . 'x arr --o 'x s arr * 'x s arr`. However, given the types of the primitives provided, these aliases can only be read from and not written to. If you want to write to this array, you must `unshare: 'x . 'x s arr --o 'x s arr --o 'x arr` it until you are left with a value of type `z arr`, guaranteeing no other aliases exists.

Given this set-up, we now *statically* have *perfect* information about aliasing and ownership of values in the program. We can only write to an array when we own it; ownership guarantees no other aliases exist in scope at the point of usage. In the next section, I show how this perfect information can be used to write more natural-looking code using value-semantic expressions which behave in precisely the way we intend it to. Now the programmer need not resort to manually tracking and inserting `noalias` annotations; instead they can let the loyal and tireless compiler do the heavy lifting.

3.3 Matrix Expressions

We have now arrived at the titular *applications* of linear types. I will show how, in addition to preventing the errors explained in the previous section, we can take linear types one step further and apply it to the domain of efficient compilation of matrix-expressions.

In Figure 3.2, we see the difficulty of efficiently implementing a *Kalman filter*, a powerful and set of equations applicable to a wide variety of problems. From the comments, we see that every variable is annotated with the step/matrix expression that it will hold at some point during the computation (an equivalent alternative, say in C++, could be to have a meaningful name for each step/matrix expression and manually annotate/keep track of which names alias the same location).

In contrast, Figure 3.3, offers the advantages of

- aliasing: labelling each step with a different, more meaningful variable name,
- easily spotting which resources are being passed in and which are allocated for the function (new/copy),
- unambiguously seeing *when* and what values are freed;

and have the compiler automatically ensure the safety of each of the above by respectively

- making it impossible to refer to steps/values which are no longer usable,
- ensuring all values are declared and *initialised* correctly before they are used,
- checking no values are used after they are freed *and* ensuring no values are leaked.

Indeed, an inexperienced programmer could take the naïve approach of just copying sub-expressions by default and then letting the compiler tell it which copies are never used and removing them systematically until it type checks. While it is not quite a black-box, push-button compilation of an expression, I would argue that, it is just as easy (if not easier) to become familiar with as Rust and its borrow-checker.

3.3.1 TO DO: Elaboration

So-called 'matrix *expressions*' are still 'side-effecting'/consuming linear values, and do not produce *values* but a *sequence of (re-)bindings* which in turn dictate what values are still available/in-scope after the computation. As such, compilation cannot be done purely compositionally via structural-induction on the expression language; it is most concisely expressed via CPS, where the result of elaborating a matrix expression is a function that takes as its argument the rest of the computation expecting to use the result it has just made available under that requested name.

3.3.2 TO DO: Inference

3.4 TO DO: Code Generation

High-level overview: primitives, translations, optimisations, difficulty, build system.

3.4.1 TO DO: Compiling Constructs to OCaml

3.4.2 TO DO: Metaprogramming

3.5 TO DO: Theory

Fractional permissions, not proven soundness etcḃut compilation preserves types as checked by OCaml.

```
subroutine f(mu, Sigma, H, INFO, R, Sigma_2, data, mu_2, k, n)
implicit none

integer, intent(in) :: k
integer, intent(in) :: n
real*8, intent(in) :: Sigma(n, n)
real*8, intent(in) :: H(k, n)
real*8, intent(in) :: mu(n)
real*8, intent(in) :: R(k, k) ! R, H*Sigma*H' + R
! (H*Sigma*H' + R)^-1*((-1)*data + H*mu), data, (-1)*data + H*mu
real*8, intent(in) :: data(k)
integer, intent(out) :: INFO
! mu, Sigma*H'*(H*Sigma*H' + R)^-1*((-1)*data + H*mu) + mu
real*8, intent(out) :: mu_2(n)
! Sigma, (-1)*Sigma*H'*(H*Sigma*H' + R)^-1*H* Sigma + Sigma
real*8, intent(out) :: Sigma_2(n, n)
real*8 :: Sigma_H(n, k) ! Sigma*H', 0
real*8 :: H_2(k, n) ! (H*Sigma*H' + R)^-1*H, H
! 0, H'*(H*Sigma*H' + R)^-1*((-1)*data + H*mu)
real*8 :: tmp_3(n)
real*8 :: tmp_2(n, n) ! 0, H'*(H*Sigma*H' + R)^-1*H
real*8 :: zeros(n, n)  ! 0
real*8 :: tmp_1(n, n) ! H'*(H*Sigma*H' + R)^-1*H*Sigma, 0

call dsymm('L', 'U', n, k, 1, Sigma, n, H, k, 0, Sigma_H, n) ! dimensions? H'? dge
call dgemm('N', 'N', k, k, n, 1, H, k, Sigma_H, n, 1, R, k)
call dcopy(n, mu, 1, mu_2, 1)
call dcopy(n**2, Sigma, 1, Sigma_2, 1)
call dgemm('N', 'N', k, 1, n, 1, H, k, mu, n, -1, data, k)
call dcopy(k*n, H, 1, H_2, 1)
call dposv('U', k, n, R, k, H_2, k, INFO)
call dposv('U', k, 1, R, k, data, k, INFO)
call dcopy(n**2, zeros, 1, tmp_2, 1)
call dgemm('N', 'N', n, n, k, 1, H, k, H_2, k, 0, tmp_2, n)
call dgemm('N', 'N', n, 1, k, 1, H, k, data, k, 0, tmp_3, n)
call dcopy(n**2, zeros, 1, tmp_1, 1)
call dsymm('L', 'U', n, n, 1, tmp_2, n, Sigma, n, 0, tmp_1, n)
call dsymm('L', 'U', n, 1, 1, Sigma, n, tmp_3, n, 1, mu_2, n)
call dsymm('L', 'U', n, n, -1, Sigma_2, n, tmp_1, n, 1, Sigma_2, n)

RETURN
END
```

Figure 3.2 – Kalman filter in Fortran 90 (Credit: Matthew Rocklin [1]).

```
let !kalman
        ('s) (sigma : 's mat) (* n, n *)
        ('h) (h : 'h mat)     (* k, n *)
        ('m) (mu : 'm arr)    (* n    *)
        (r_1 : z mat)         (* k, k *)
        (data_1 : z arr)      (* k    *)
        : ('s mat * ('h mat * 'm arr)) * (z arr * z mat) =
    let ((!k, !n), h) = dimM _ h in
    let zeros = zeroM n n in
    let sigma_h = new [| sym (up, sigma) * h |] in
    let r_2 = [| h * sigma_h + r_1 |] in
    let mu_1 <- copy mu in
    let sigma_1 <- copyM sigma in
    let data_2 = [| h * mu + -1. * data_1 |] in
    let h_1 <- copyM h in
    let h_sol = [| sym (up, r) ^ -1 * h_1 ] in
    let data_sol = [| sym (up, r) ^ -1 * data_2 |] in
    let h_h1 = new [| h * h_sol |] in
    let h_data = new [| h * data |] in
    let h_h1_sigma = new [| sym (up, h_h1) * sigma |] in
    let mu_2 = [| sym (up, sigma) * h_data * mu_1 |] in
    (* non-linear use of sigma_c..? *)
    let sigma_c = [|  -1. * sym (up, sigma_c) * h_h1_sigma + sigma_c |] in
    (* frees... *)
    (mu_c , sigma_c)
```

Figure 3.3 – Kalman filter in LT4LA.

4 | TO DO: Evaluation

For any practical projects, you should almost certainly have some kind of evaluation, and it's often useful to separate this out into its own chapter.

In this chapter, I will argue the central premise of this thesis: linear types are a practical and usable tool to help working programmers write code that is both more legible and less resource-hungry than with existing linear-algebra frameworks. My project illustrates how to do so in a way that can be implemented as a usable *library* for existing languages and frameworks that leverages the already impressive amount of work gone into optimising them so far.

4.1 Set-up

4.2 Results

4.3 Summary

5 | Related Work

Now that I have described my contributions, I will explain how it relates to existing work, leaving brief discussions on future work to the next chapter. I strongly believe matrix expression compilation research would benefit greatly from a comprehensive literature review but unfortunately that is beyond the scope of this chapter.

5.1 Matrix Expression Compilation

Most of the projects below try to be fully-automated black-boxes which model computing a matrix expression as some sort of graph with informal, ad-hoc rules about what can and should be copied or modified in-place. Allocations, temporaries and common sub-expressions are details invisible to the programmer, left to the compiler.

The matrix expression 'compiler' as implemented in LT4LA is intended to be a mere proof-of-concept of how linear types can and arguably should be used *ergonomically*. I have taken the approach of attempting to help programmers precisely and explicitly capture, using types, the practices prevalent in code they already write.

I believe the advantages of my approach are two-fold (1) *predictable* performance and (2) more accurate modelling of how low-level kernels handle their resources. My confidence in the latter claim comes from finding at least two errors in the Fortran code output by SymPy to compute a Kalman filter, one to do with aliasing (resulting in code that was not legal Fortran) that would not have type-checked had it been translated via LT4LA as an intermediate representation (the other error to be explained later in 5.3.5).

5.1.1 SymPy

SymPy is a symbolic computer algebra system for Python; its matrix expression compiler [6] uses a term-rewrite system, with rules supplied by a BLAS expert (which must be strongly-normalising, that is, never cause a loop) but need not be confluent (there can be more than one solution per expression). Rules include expressions to match on, the expression it can produce, information about the expressions (such as whether the matrix is symmetric or full-rank) and information about which variable is updated in-place.

5.1.2 Clak and Cl1ck

Clak and Cl1ck [7] were developed independently around the same time as SymPy's matrix expression compilation. Clak attempts to produce *multiple* algorithms for a single matrix expression, by considering a wider matrix expression grammar

and more matrix properties and inference rules. These algorithms assume basic building blocks such as products and factorisations. Cl1ck attempts to take on the challenge of writing BLAS/LAPACK like libraries too, by generating lower-level loop-based blocked routines for the aforementioned basic building blocks, in the spirit of the FLAME [8] project.

5.1.3 Linnea and Taco

Linnea [9] and Taco [10] are two newer contenders to Clak and Cl1ck respectively. Linnea continues the work of Clak to producing real executable code for *existing* libraries and kernels, as well as incorporating work on a *generalised* matrix chain algorithm [11]. Taco (*Tensor Algebra COmpiler*) focuses on emitting efficient routines for expressions in tensor index notation, with many optimisations for *sparse* tensors.

5.2 Metaprogramming

Most of the compilers in the aforementioned projects usually built, analysed, compiled, ran (and in some cases, dynamically loaded) expressions (including functions) at runtime, similar to how regular expressions are handled in most languages – in particular, even when the regular expression is known at compile time.

In LT4LA, I took the approach of having a concrete syntax and expression language which was then translated and made available as a *typed expression* to other modules *at compile time* via the build system. Apart from convenience in programming and testing, there was nothing inherent in the approach that prevented me from using OCaml's PPX syntax-extensions so that I could write normal OCaml expressions from within OCaml and have them checked for linearity before compilation.

Having a statically compiled language and a build system as so affords the advantage of eliminating the runtime overheads mentioned at the start of this section. However, there is some useful information (such as matrix dimensions for the matrix chain algorithm) which is sometimes known only at runtime (but once known, usually fixed). In these cases, using *multi-stage programming* would be a better approach to implementing a matrix expression compiler.

5.2.1 MetaOCaml and Scala LMS

MetaOCaml [12] and Scala with Lightweight Modular Staging [13] are systems which support multi-stage programming. A typical example of this is the generation of Fast Fourier-Transform kernels, specialised to a desired array length.

5.2.2 Expression Templates in C++ Libraries

Expression templates are a commonly used compile-time metaprogramming technique, used by Eigen, uBLAS and Armadillo to name a few. If known at compile-time, matrix dimensions can also be passed in as template arguments to ensure operations match (otherwise checking at runtime). In Eigen, such features are combined with heuristics to enable *lazy evaluation* and automatically determine whether a sub-expression is evaluated into a temporary variable or not.

They perform rudimentary pattern-matching and in some cases, loop-fusion, to avoid evaluating expressions in a purely binary manner (invoking the bane of a C++ programmer: temporaries and unnecessary copies) when possible (either by translating to a library kernel call or, as an example, inlining a $v := a + b + c$ vector expression into one loop).

As is the case with LT4LA, this approach shares *some* elimination of runtime overheads, but not all, thanks to the heuristics surrounding lazy evaluation and evaluating sub-expressions into temporaries. This comes at the cost of a user being able to easily inspect the generated C++ code, losing explicitness.

5.3 Types

Apart from lazy evaluation, the following projects show how instead of a (E)DSL library approach, we could have type-level resource management provided far more conveniently and naturally at the *language* level. The difference is that a library can be designed, shipped and used now whereas language features take time and can have unintended interactions with other language features. My hope is that once people are convinced of the utility type-level resource management by using a library, the impetus for integrating such features into the language follows.

5.3.1 Lazy Evaluation

A particularly strong advantage LT4LA has over other libraries that use lazy evaluation is, funnily enough, linear types, more precisely and the static and perfect information they guarantee about aliasing: with Owl, every graph-node has only one node in or out; with Eigen, every "assignment" has a `noalias` annotation.

This simplifies the rules and exceptions a programmer reasoning about memory usage needs to remember. Of course, now the programmer has to figure out how they are using their temporaries, but because matrices are linearly-typed, redundant copies/missed frees can be pointed out by the compiler, guiding them towards a satisfying solution.

5.3.2 Futhark

Futhark [14] is a second-order (meaning it supports functions such as map and fold/reduce) array combinator (meaning array operations can be fused into streams to reduce temporaries) language designed for efficient parallel compilation. It supports ML-style modules, loops, limited parametric polymorphism, size types and uniqueness types. Intuitively, where linearity provides a local guarantee of "this value is not aliased in this scope", uniqueness types provide a global guarantee of "this value is not aliased anywhere". Its combinators are more expressive than typical linear algebra library kernels so encourages shorter, more declarative linear algebra code.

5.3.3 Substructural Features in Rust

Rust [15] is a (relatively) new systems programming language aiming to bring the last two decades of programming language research to the masses in a usable and friendly manner. Its *borrow-checker* is the feature most relevant to this thesis because it statically attempts to prevent many resource-related bugs. Although there are a few linear algebra libraries for Rust under development, careful use of its macro system and borrow-checker could make it the safest and easiest to use language for linear algebra projects to come. Its struggle is more likely to be against the inertia of the large amounts of C++/Fortran code already out there rather than its usability or benefits.

5.3.4 Linear Types in Haskell

Linear types have been incorporated into a branch of the Glasgow Haskell Compiler [16] in an attempt to provide safe, functional streaming and IO (after people saw the potential from libraries providing linearity features). Practical benefits include zero-copy buffers and eliminating garbage collection in certain situations by allowing the user to safely manage memory. The fact that it can and has been done gives me hope that other languages will also see the value and adopt some form of resource-management in their type systems.

5.3.5 Linear and Dependent Types in Idris

Dimension mismatches are seen as an irritating but small inconvenience when writing linear algebra code. However, the second error I found in the Fortran code output by SymPy to compute a Kalman filter was a dimension/transposition error. Although we would not need full dependent types to solve dimension mismatches (symbolic size types would be sufficient), managing properties about matrices could be done at the type-level in a dependently typed setting.

We could then express the usual properties and results of operations at the type-level, ensuring, for example, that certain functions are called only when the matrix is symmetric and can be written to. Idris (a Haskell inspired language with dependent types) has had experimental support for uniqueness types since its early days and now a linear types extension [17] is also being worked on based on new research around integrating linear and dependent types [5].

6 | TO DO: Conclusion

As you might imagine: summarizes the dissertation, and draws any conclusions. Depending on the length of your work, and how well you write, you may not need a summary here.

You will generally want to draw some conclusions, and point to potential future work.

What are the benefits of the type-based approach I have taken? Have I successfully argued whatever I wrote in the introduction?

6.1 Future Work

Some directions in which a type-based approach to efficient matrix expression compilation could be taken are:

- **As a typed IR** for matrix expression compilers. This in turn could enable
  - existing matrix expression compilers to be less opaque about what resources they are consuming.
  - open up opportunities for non-local sharing of temporary values with some intra-procedural analysis.
  - allow the user to choose: use a matrix expression compiler when desired and drop down to a usable typed-IR for finer control, whilst still retaining safety guarantees.
- **Formal verification of matrix expression compilers** by precisely specifying source and target languages.

- **Multi-stage programming** to use information only available at runtime in many situations (such as sizes, matrix properties, sparsity, control flow) can be effectively incorporated into code generated.
- **Dependent types** to have control over how resources can be used and split. In addition to formal verification, dependent types could be combined with linearity to express finer-grain conventions surrounding blocking, slicing and writing to *parts* of the matrix instead of the whole. This is already prevalent with 'dsymm' like BLAS routines which only read and write to the lower or upper triangle of a matrix. This idea is inspired by Conor McBride's talk on writing to terminals with "Space Monads" [18].
- **Compiling to hardware** is also an option - once we know exactly when and where temporaries are required and what can be re-used when, we come one (small, but useful) step closer to realising matrix expressions directly on hardware.

6.1.1 Links to Linearity

All of the matrix expression compilers mentioned in previous section construct some sort of dataflow graph to represent the computation being executed. While this seems intuitive, there is no formal argument for this appr

A | Ott Specification

The following pages present a specification of the grammar and type-system used by my project, produced using the Ott[19] tool. It is important to note that the type-system described here is not how it is implemented: it is easier and clearer to describe the system as below for explaining. However, for implementing, I found it much more and user- and debugging-friendly to:

- Implement it so that the type-environment *changes* as a result of type-checking an expression; with this, the below semantics describe the *difference* between the environment after and before checking an expression.
- *Mark* variables as used instead of *removing* them from the environment for better error messages.
- Have *one* environment where variables were *tagged* as linear and un-used, linear and used, and intuitionistic. This was definitely an implementation convenience so that variable binding could be handled uniformly for linear and intuitionistic variables and scoping/variable look-up could be handled implicitly thanks to the associative-list structure of the environment.

fc	fractional capability variable
x, g, a, b	expression variable
k	integer variable
el	array-element variable

symb	::=		
	|	λ	
	|	$\otimes$	
	|	$\multimap$	
	|	$\vdash$	
	|	$\in$	
	|	$\forall$	
	|	Cap	
	|	Type	
	|	!	
	|	$\rightarrow$	
	|	**value**	

f	::=			fractional capability
	|	fc		variable
	|	**Z**		zero
	|	**S** f		successor

t	::=			linear type
	|	**unit**		unit
	|	**bool**		boolean (true/false)
	|	**int**		63-bit integers
	|	**elt**		array element
	|	f **arr**		arrays
	|	f **mat**		matrices
	|	$!t$		multiple-use type
	|	$\forall\, fc.t$	bind fc in t	frac. cap. generalisation
	|	$t \otimes t'$		pair
	|	$t \multimap t'$		linear function
	|	$t\{f/fc\}$	M	substitution
	|	(t)	S	parentheses

e	::=		expression
	|	p	primitives (arithmetic, L1 BLAS, Owl)
	|	x	variable

	$()$		unit introduction
|	$\mathbf{true}$		true (boolean introduction)
|	$\mathbf{false}$		false (boolean introduction)
|	$\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$		if (boolean elimination)
|	k		integer
|	el		array element
|	$\mathbf{many}\ e$		packing-up a non-linear value
|	$\mathbf{let\,many}\ x = e\ \mathbf{in}\ e'$		using a non-linear value
|	$\mathbf{fun}\ fc \to e$		frac. cap. abstraction
|	$e[f]$		frac. cap. specialisation
|	(e, e')		pair introduction
|	$\mathbf{let}\ (a, b) = e\ \mathbf{in}\ e'$	$\mathsf{bind}\ a \cup b\ \mathsf{in}\ e'$	pair elimination
|	$\mathbf{fun}\ x : t \to e$	$\mathsf{bind}\ x\ \mathsf{in}\ e$	abstraction
|	$e\ e'$		application
|	$\mathbf{fix}\ (g, x : t, e : t')$	$\mathsf{bind}\ g \cup x\ \mathsf{in}\ e$	fixpoint

p	::=		primitive
	| $\mathbf{set}$		array index assignment
	| $\mathbf{get}$		array indexing
	| $(+)$		integer addition
	| $(-)$		integer subtraction
	| $(*)$		integer multiplication
	| $(/)$		integer division
	| $(=)$		integer equality
	| $(<)$		integer less-than
	| $(+.)$		element addition
	| $(-.)$		element subtraction
	| $(*.)$		element multiplication
	| $(/.)$		element division
	| $(= .)$		element equality
	| $(< .)$		element comparsion (less-than)
	| $(\&\&)$		boolean conjuction
	| $(\|\|)$		boolean disjunction
	| $\mathbf{not}$		boolean negation

	share	share array
	unshare	unshare array
	free	free arrary
	array	Owl: make array
	copy	Owl: copy array
	sin	Owl: sine of all elements in array
	hypot	Owl: $x_i := \sqrt{x_i^2 + y_i^2}$
	asum	BLAS: $\sum_i \lvert x_i \rvert$
	axpy	BLAS: $x := \alpha x + y$
	dot	BLAS: $x \cdot y$
	rotmg	BLAS: gen. mod. Givens rotation
	scal	BLAS: $x := \alpha x$
	amax	BLAS: index of maximum absolute value
	setM	matrix index assignment
	getM	matrix indexing
	shareM	share matrix
	unshareM	unshare matrix
	freeM	free matrix
	matrix	Owl: make matrix
	copyM	Owl: copy matrix
	gemv	BLAS: $y := \alpha A^{T?} x + \beta y$
	symv	BLAS: $y := \alpha A_{\text{sym}} x + \beta y$
	trmv	BLAS: $x := A^{T?} * x$
	trsv	BLAS: $x := A^{-1 \cdot T?} * x$
	ger	BLAS: $A := \alpha * x * y^T + A$
	gemm	BLAS: $C := \alpha * A^{T?} * B^{T?} + \beta C$
	trmm	BLAS: $B := \alpha * A^{T?} * B$ and swapped
	trsm	BLAS: $B := \alpha * A^{-1 \cdot T?} * B$ and swapped

Θ	::=		fractional capability environment
	|	$\cdot$	
	|	Θ, fc	

Γ	::=		linear types environment

	$\mid$	$\cdot$
	$\mid$	$\Gamma, x : t$
	$\mid$	Γ, Γ'
Δ	::=	linear types environment
	$\mid$	$\cdot$
	$\mid$	$\Delta, x : t$
formula	::=	
	$\mid$	*judgement*
	$\mid$	$x : t \in \Delta$
	$\mid$	$x : t \in \Gamma$
	$\mid$	$fc \in \Theta$
	$\mid$	$\mathbf{value}\,(e)$
Well_Formed	::=	
	$\mid$	$\Theta \vdash f\ \mathsf{Cap}$ Valid fractional capabilities
	$\mid$	$\Theta \vdash t\ \mathsf{Type}$ Valid types
Values	::=	
	$\mid$	$\mathbf{value}\,(e)$ Value restriction for !-introduction
Types	::=	
	$\mid$	$\Theta; \Delta; \Gamma \vdash e : t$ Typing rules for expressions (no primitives yet)
judgement	::=	
	$\mid$	*Well_Formed*
	$\mid$	*Values*
	$\mid$	*Types*
user_syntax	::=	
	$\mid$	*fc*
	$\mid$	*x*
	$\mid$	*k*
	$\mid$	*el*
	$\mid$	*symb*

| f
| t
| e
| p
| Θ
| Γ
| Δ
| *formula*

$\boxed{\Theta \vdash f\ \mathsf{Cap}}$ Valid fractional capabilities

$$\frac{fc \in \Theta}{\Theta \vdash fc\ \mathsf{Cap}}\ \textsc{WF\_Cap\_Var}$$

$$\frac{}{\Theta \vdash \mathbf{Z}\ \mathsf{Cap}}\ \textsc{WF\_Cap\_Zero}$$

$$\frac{\Theta \vdash f\ \mathsf{Cap}}{\Theta \vdash \mathbf{S}\, f\ \mathsf{Cap}}\ \textsc{WF\_Cap\_Succ}$$

$\boxed{\Theta \vdash t\ \mathsf{Type}}$ Valid types

$$\frac{}{\Theta \vdash \mathbf{unit}\ \mathsf{Type}}\ \textsc{WF\_Type\_Unit}$$

$$\frac{}{\Theta \vdash \mathbf{bool}\ \mathsf{Type}}\ \textsc{WF\_Type\_Bool}$$

$$\frac{}{\Theta \vdash \mathbf{int}\ \mathsf{Type}}\ \textsc{WF\_Type\_Int}$$

$$\frac{}{\Theta \vdash \mathbf{elt}\ \mathsf{Type}}\ \textsc{WF\_Type\_Elt}$$

$$\frac{\Theta \vdash f\ \mathsf{Cap}}{\Theta \vdash f\ \mathbf{arr}\ \mathsf{Type}}\ \textsc{WF\_Type\_Array}$$

$$\frac{\Theta \vdash t\ \mathsf{Type}}{\Theta \vdash !t\ \mathsf{Type}}\ \textsc{WF\_Type\_Bang}$$

$$\frac{\Theta, fc \vdash t\ \mathsf{Type}}{\Theta \vdash \forall fc.t\ \mathsf{Type}}\ \textsc{WF\_Type\_Gen}$$

$$\frac{\Theta \vdash t\ \mathsf{Type} \quad \Theta \vdash t'\ \mathsf{Type}}{\Theta \vdash t \otimes t'\ \mathsf{Type}}\ \textsc{WF\_Type\_Pair}$$

$$\frac{\Theta \vdash t\ \mathsf{Type} \quad \Theta \vdash t'\ \mathsf{Type}}{\Theta \vdash t \multimap t'\ \mathsf{Type}}\ \textsc{WF\_Type\_Lolly}$$

$\boxed{\mathbf{value}\,(e)}$ Value restriction for !-introduction

$$\frac{}{\mathbf{value}\,(p)}\ \textsc{Val\_Prim}$$

$$\frac{}{\mathbf{value}\,(())}\ \textsc{Val\_Unit\_Intro}$$

$$\frac{}{\mathbf{value}\,(\mathbf{true})}\ \textsc{Val\_Bool\_True}$$

$$\frac{}{\mathbf{value}\,(\mathbf{false})}\ \textsc{Val\_Bool\_False}$$

$$\frac{}{\mathbf{value}\,(k)}\ \textsc{Val\_Int\_Intro}$$

$$\frac{}{\mathbf{value}\,(el)}\ \textsc{Val\_Elt\_Intro}$$

$$\frac{}{\mathbf{value}\,(x)}\ \textsc{Val\_Var}$$

$$\frac{}{\mathbf{value}\,(\mathbf{fix}\,(g, x : t, e : t'))}\ \textsc{Val\_Fix}$$

$$\frac{}{\mathbf{value}\,(\mathbf{fun}\,x : t \to e)}\ \textsc{Val\_Lambda}$$

$$\frac{\mathbf{value}\,(e)}{\mathbf{value}\,(\mathbf{fun}\,fc \to e)}\ \textsc{Val\_Gen}$$

$$\frac{\mathbf{value}\,(e)}{\mathbf{value}\,(e[fc])}\ \textsc{Val\_Spc}$$

$$\frac{\mathbf{value}\,(e)}{\mathbf{value}\,(\mathbf{many}\,e)}\ \textsc{Val\_Bang\_Intro}$$

$$\frac{\mathbf{value}\,(e_1) \quad \mathbf{value}\,(e_2)}{\mathbf{value}\,((e_1, e_2))}\ \textsc{Val\_Pair\_Intro}$$

$\boxed{\Theta; \Delta; \Gamma \vdash e : t}$ Typing rules for expressions (no primitives yet)

$$\frac{}{\Theta; \Delta; \cdot, x : t \vdash x : t}\ \textsc{Ty\_Var\_Lin}$$

$$\frac{x : t \in \Delta}{\Theta; \Delta; \cdot \vdash x : t}\ \textsc{Ty\_Var}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash () : !\mathbf{unit}} \quad \textsc{Ty\_Unit\_Intro}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash \mathbf{true} : !\mathbf{bool}} \quad \textsc{Ty\_Bool\_True}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash \mathbf{false} : !\mathbf{bool}} \quad \textsc{Ty\_Bool\_False}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash e : \mathbf{bool} \\ \Theta; \Delta; \Gamma' \vdash e_1 : t' \\ \Theta; \Delta; \Gamma' \vdash e_2 : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t} \quad \textsc{Ty\_Bool\_Elim}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash k : !\mathbf{int}} \quad \textsc{Ty\_Int\_Intro}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash el : !\mathbf{elt}} \quad \textsc{Ty\_Elt\_Intro}$$

$$\frac{\begin{array}{c}\mathbf{value}\,(e) \\ \Theta; \Delta; \cdot \vdash e : t\end{array}}{\Theta; \Delta; \cdot \vdash \mathbf{many}\ e : !t} \quad \textsc{Ty\_Bang\_Intro}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash e : !t \\ \Theta; \Delta, x : t; \Gamma' \vdash e' : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let\,many}\ x = e\ \mathbf{in}\ e' : t'} \quad \textsc{Ty\_Bang\_Elim}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash e : t \\ \Theta; \Delta; \Gamma' \vdash e' : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash (e, e') : t \otimes t'} \quad \textsc{Ty\_Pair\_Intro}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash e_{12} : t_1 \otimes t_2 \\ \Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e : t\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let}\,(a, b) = e_{12}\,\mathbf{in}\ e : t} \quad \textsc{Ty\_Pair\_Elim}$$

$$\frac{\begin{array}{c}\Theta \vdash t' \,\mathsf{Type} \\ \Theta; \Delta; \Gamma, x : t' \vdash e : t\end{array}}{\Theta; \Delta; \Gamma \vdash \mathbf{fun}\ x : t' \to e : t' \multimap t} \quad \textsc{Ty\_Lambda}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash e : t' \multimap t \\ \Theta; \Delta; \Gamma' \vdash e' : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash e\ e' : t} \quad \textsc{Ty\_App}$$

$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun}\, fc \to e : \forall\, fc.t} \quad \textsc{Ty\_Gen}$$

$$\frac{\begin{array}{c}\Theta \vdash f\ \mathsf{Cap} \\ \Theta; \Delta; \Gamma \vdash e : \forall\, fc.t\end{array}}{\Theta; \Delta; \Gamma \vdash e[f] : t\{f/fc\}}\ \textsc{Ty\_Spc}$$

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \mathbf{fix}\,(g, x : t, e : t') : !(t \multimap t')}\ \textsc{Ty\_Fix}$$

Bibliography

[1] Matthew Rocklin. Kalman filter in blas/lapack fortran. https://gist.github.com/mrocklin/5144149#file-kalman-f90.raw, 2018.

[2] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[3] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In *International Colloquium on Automata, Languages, and Programming*, pages 385–397. Springer, 2013.

[4] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *International Conference on Computer Aided Verification*, pages 781–786. Springer, 2012.

[5] Robert Atkey. The syntax and semantics of quantitative type theory.(2017). *Under submission*, 2017.

[6] Matthew Rocklin. *Mathematically informed linear algebra codes through term rewriting*. PhD thesis, 2013.

[7] Diego Fabregat-Traver. Knowledge-based automatic generation of linear algebra algorithms and code. *arXiv preprint arXiv:1404.3406*, 2014.

[8] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001.

[9] Henrik Barthels and Paolo Bientinesi. Linnea: Compiling linear algebra expressions to high-performance code. In *Proceedings of the 8th International Workshop on Parallel Symbolic Computation*, Kaiserslautern, Germany, July 2017.

[10] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[11] Henrik Barthels, Marcin Copik, and Paolo Bientinesi. The generalized matrix chain algorithm. *arXiv preprint arXiv:1804.04021*, 2018.

[12] Oleg Kiselyov. The design and implementation of ber metaocaml. In *International Symposium on Functional and Logic Programming*, pages 86–102. Springer, 2014.

[13] Tiark Rompf. Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming. *ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE*, 2012.

[14] Troels Henriksen. Design and implementation of the futhark programming language (revised). 2017.

[15] Rust Community. Rust. https://www.rust-lang.org/en-US. Accessed: 08/05/2018.

[16] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Retrofitting linear types, 2017.

[17] Idris Community. Idris 1.2.0 release notes. https://www.idris-lang.org/idris-1-2-0-released/. Accessed: 08/05/2018.

[18] Conor McBride. Code mesh london 2016, keynote: Spacemonads. https://www.youtube.com/watch?v=QojLQY5HORI. Accessed: 08/05/2018.

[19] Ott. http://www.cl.cam.ac.uk/~pes20/ott/. Accessed: 29/04/2018.